Analisis y Diseño de Algoritmos

Juan Miguel Rojas

15 de abril de 2024

1. Ejercicio 4-17: Interval Scheduling Problem (Kleinberg & Tardos, página 197).

Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if the job is accepted to run on the processor, it must run conl nuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)
Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.
Example. Consider the fol!owing four jobs, specified by (start-time, endtime) pairs.

(6 P.M., 6 A.M.), (9 P.M., 4 A.M.), (3 A.M., 2 P.M.), (1 P.M., 7 P.M.).

The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

Solución

Entrada: Una lista de tuplas $A[0...n]$ con el horario de inicio y el horario de finalización sin que se repitan procesos con la misma hora inicio u hora de finalización.
Salida: La cantidad maxima de procesos sin que se solapen.

Podemos tomar los procesos **(Figura 1)** y ver que se parecen a un reloj del día a día, pero para efectos del ejercicio debemos tomar las horas en horario militar ya que esto tiene un factor importante.

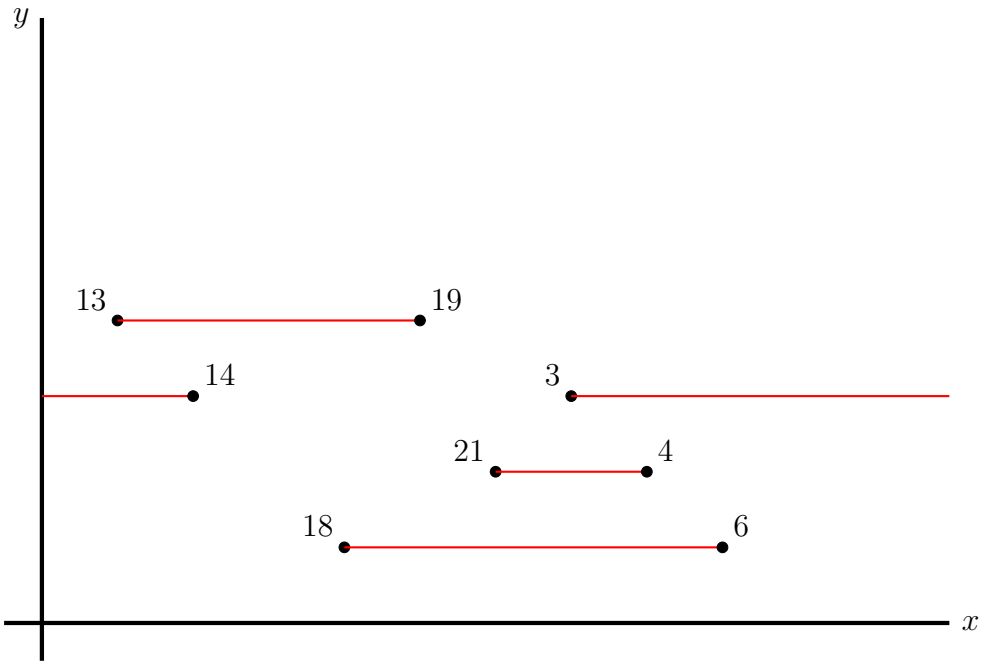

Figura 1: **Ejemplo grafico**

Estrategia voraz: La estrategía voraz que utilizaremos será seleccionar los procesos por duración y las ordenaremos por esto mismo **(Figura 2)**, para efectos del problema tendremos que cambiar algunos horarios a lineales, es decir, si tengo un horario que sobrepasa la medianoche, deberemos sumarle 24 horas.

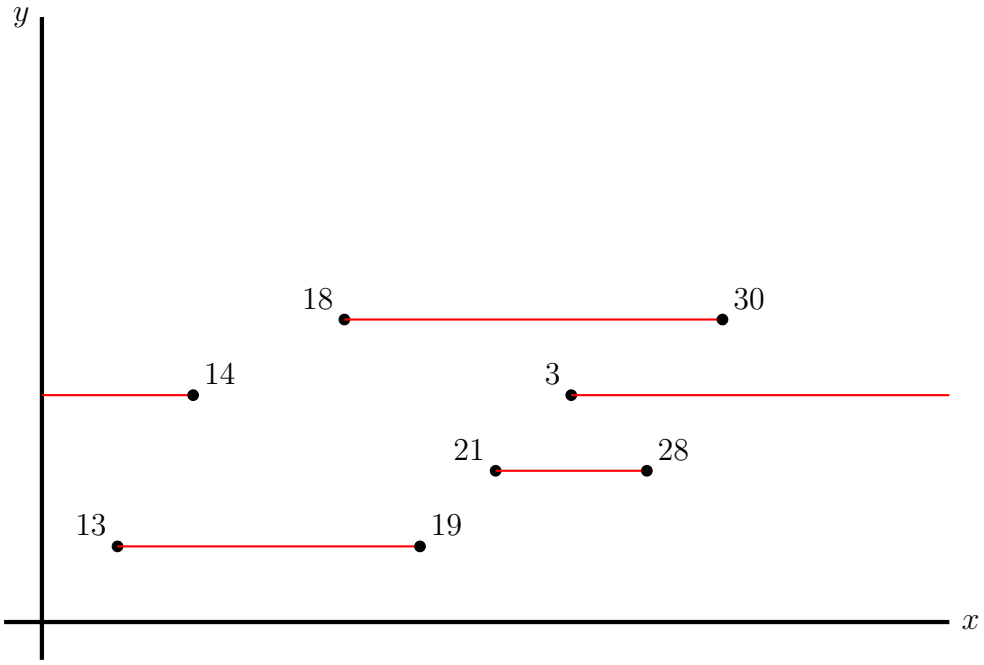

Figura 2: **Procesos ordenados por duración**

De este modo tendremos organizado los procesos por duración, y aplicaremos la estrategia voraz y los resultados serían (13, 19) y (21, 28) ya que son los horarios que no se solapan **(Figura 3)** y estos serían los horarios respectivamente (1 P.M., 7 P.M.) y (9 P.M., 4 A.M.).

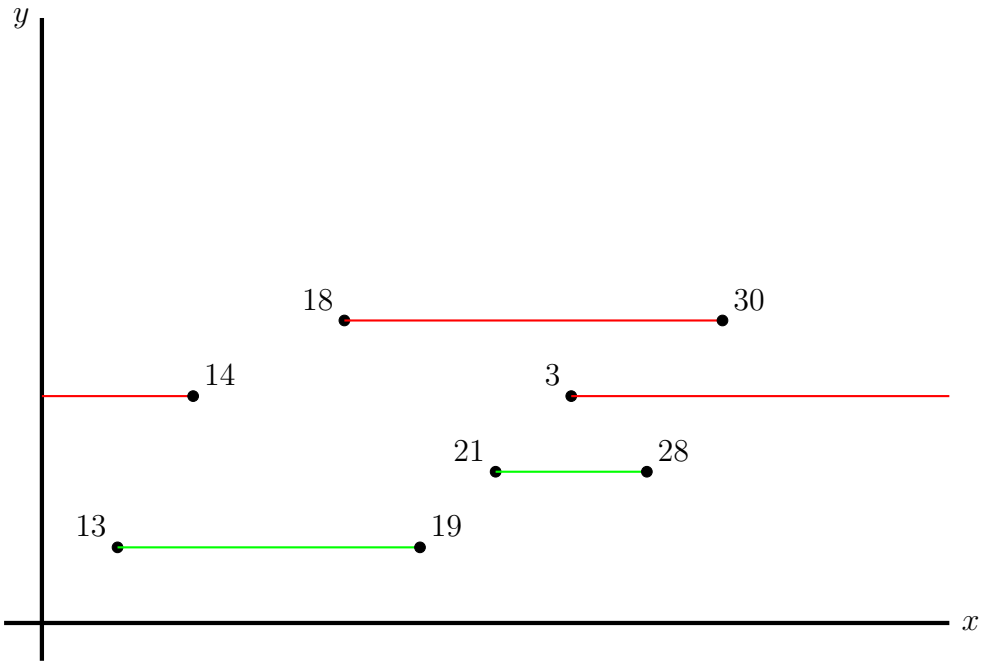

Figura 3: **Estrategía voraz**

Teorema de optimización local

Sea A un conjunto de procesos y sea a_m el proceso con menor duración en el conjunto de procesos A, entonces tenemos que a_m hace parte de un conjunto el cual contiene la maxima cantidad de procesos sin que se solapen.

Demostración:

Sea $B \subseteq A$ un conjunto de procesos sin que se solapen y sea la mayor cantidad. Y sea b_m el proceso que tiene menor duración en B. Por lo tanto, existen 2 posibilidades:

- $a_m = b_m$ y en consecuencia a_m pertenece al conjunto B. En este caso a_m pertenece al conjunto resultante B del teorema.
- $a_m \neq b_m$ Entonces demostrare que a_m pertenece a un conjunto con la mayor cantidad de procesos sin solaparse. Consideramos un conjunto $B' = B - \{b_m\} \cup \{a_m\}$, con esto podemos darnos cuenta de que B' no tiene conflicto con A, debido el tiempo de duración a_m es minimo en A, por ende, también en B'. Ademas podemos saber que $|B'| = |B|$, por lo cual, B' es optimo. Esto debido a que sabemos que $a_m \in B'$, entonces hace parte de un conjunto con la mayor cantidad de procesos y sin solaparse con los procesos del conjunto A

Función objetivo

Sea $0 \leq i \leq N$

$\phi(i)$: Maxima cantidad de procesos en $A[i...N]$ que empiezan después del intervalo, pueden ser agendadas sin solapamiento y que cumplan el ciclo de la medianoche.

Reformulación de la especificación

Entrada: Una lista de tuplas $A[0...n]$ con el horario de inicio y el horario de finalización sin que se repitan procesos con la misma hora inicio u hora de finalización.

Salida: $phi(A)$.

Planteamiento recurrente

Mi planteamineto recurrente cuenta con dos funciones, mi función principal se encarga de seleccionar procesos y mi función auxiliar se encarga de realizar la verificación de que no exista solapamiento entre los procesos ya seleccionados.
Primero definire mi funcion auxiliar, la cual recibira los procesos ya seleccionados y un proceso a verificar si se solapa.

- Condición: $(\textbf{lans[j][0]} \geq \textbf{schedule[1]} \wedge \textbf{lans[j][0]} > \textbf{schedule[0]}) \vee (\textbf{lans[j][1]} \leq \textbf{schedule[0]} \wedge \textbf{lans[j][1]} < \textbf{schedule[1]})$

Sea $lans[0...M]$ y $0 \leq M \leq N$

$$\beta(lans, sche, j) = \begin{cases} True & M = 0 \vee j = M \\ False & lans[j][1] >= 24 \wedge lans[j][0] + sche[1] \geq lans[j][1] \\ False & \neg \text{Condición} \\ \beta(lans, schedule, j+1) & \text{Condición} \end{cases}$$

$$\phi(i, lans) = \begin{cases} 0 & N = 0 \vee i = N \\ add(A[i]), 1 + \phi(A, i+1, lans) & \beta(lans, schedule) \\ \phi(A, i+1, lans) & \neg\beta(lans, schedule) \end{cases}$$

Codigo

Listing 1: Función recurrente

```
def beta(lans, schedule, j):
  ans = None
  if(j == len(lans)):
    ans = True
  elif(lans[j][1] >= 24 and lans[j][0] + schedule[1] >= lans[j][1]):
    ans = False
  elif((lans[j][0] >= schedule[1] and lans[j][0] > schedule[0]) or
     (lans[j][1] <= schedule[0] and lans[j][1] < schedule[1])):
    ans = beta(lans, schedule, j + 1)
  else:
    ans = False

  return ans

def phi(A, i, lans):
  ans = 0
  if(i == len(A)):
    ans = 0
  else:
    if(beta(lans, A[i], 0)):
      lans.append(A[i])
      ans = 1 + phi(A, i + 1, lans)
    else:
      ans = phi(A, i + 1, lans)

  return ans
```

Listing 2: Función iterativa

```
def verificar(lans, schedule):
  #print(schedule)
  ans = True
  i = 0
  if(len(lans) == 0):
    ans = True
  else:
    while(i != len(lans) and ans):
      if(lans[i][1] >= 24 and lans[i][0] + schedule[1] >= lans[i][1]):
        ans = False
      elif((lans[i][0] >= schedule[1] and lans[i][0] > schedule[0]) or
         (lans[i][1] <= schedule[0] and lans[i][1] < schedule[1])):
        ans = True
      else:
        ans = False
      i+=1

  return ans

def solve(A):
  lans = []
  ans = 0
  for schedule in A:
    if(verificar(lans, schedule)):
      ans+=1
      lans.append((schedule[0], schedule[1]))

  return lans, ans
```

Teorema de optimización global

Sea $0 \leq i \leq N$ el llamado de $\phi(i, lans)$ produce la maxima cantidad de procesos en $A[i..N)$ que pueden ser ejecutados sin solaparse.

- **Caso base ($j = M$):** En caso de que $A[i..N)$ sea vacío entonces producira 0 procesos.
- **Caso inductivo ($j \neq M$):** Por HI sabemos que $phi(j+1, lans')$ producira la mayor cantidad de procesos que se pueden agregar sin que se solapen con $lans'$, por lo cual existen dos posibilidades:
  - $\beta(lans, A[i], j) = True$: Dado que las actividades $A[0..N)$ estan ordenadas por tiempo de duración, obtenemos que $A[i]$ es la actividad con menor tiempo de duración. También sabemos que por el teorema de optimización local $A[i]$ hace parte de un agendamiento optimo y por HI sabemos que $phi(j+1, lans')$ produce una solución optima para $A[i+1...N]$ y en consecuencia produciremos $add(A[i]), 1 + \phi(i+1, lans)$, es decir, $A[i] \in lans[0...M]$.
  - $\beta(lans, A[i], j) = False$: En este caso $A[i]$ se encuentra entre alguno de los procesos de $lans[0...M]$, es decir, se solapan. Y eso lo podemos saber porque para cada $a_m \in A$ que no cumpla la Condición y esto permite saber que existe un $l_m \in lans$ que se solapa con a_m

Corolario: El llamado de $phi(0, [\,])$ produce la mayor cantidad de procesos sin que se solapen.

Complejidad

La complejidad del algoritmo es $O(N \cdot M \cdot log(N)$ si contemplamos que los datos no ingresan ordenados por tiempo de duración, si se puede considerar que los datos del problema se encuentran ordenados siempre, entonces la complejidad es $O(N \cdot M)$

Pruebas

Para esta parte tengo una serie de casos de prueba con los cuale pude verificar que mi algoritmo utilizara la estrategia correcta y produjera la mayor cantidad de actividades sin solaparse.
Por lo tanto esto es una especie de entrada, evidentemente en mi algoritmo planteado anteriormente no considero el ordenamiento, pero aquí ya ordeno los datos y calculo el tiempo de duración con la función tiempo de duración que es la siguiente.

Listing 3: Calcular el tiempo de duración

```
def tiempoDuracion(act):
  temp = []
  for a in act:
    time = a[1] - a[0]
    temp.append((a[0], a[1], time))
  return temp
```

Casos de prueba

```
def main():
  act = [(2, 25), (24, 28), (3, 23)]
  act1 = [(18, 30), (21, 28), (3, 14), (13, 19)]
  act2 = [(1, 3),(4, 20),(21, 26)]
  act3 = [(16, 20), (5, 10), (12, 35), (18, 30), (21, 28), (3,14), (13,
     19)]
  act4 = [(2, 26), (24, 28), (3, 23)]
  act5 = [(13, 14), (15, 16), (23, 37)]
  act6 = [(17, 26), (2, 3), (18, 20), (20, 22)]
  act7 = [(4,27), (10,16), (21,4)]
  act = tiempoDuracion(act)
  act1 = tiempoDuracion(act1)
  act2 = tiempoDuracion(act2)
  act3 = tiempoDuracion(act3)
  act4 = tiempoDuracion(act4)
  act5 = tiempoDuracion(act5)
  act6 = tiempoDuracion(act6)
  act7 = tiempoDuracion(act7)
  act.sort(key = lambda x: x[2])
  act1.sort(key = lambda x: x[2])
  act2.sort(key = lambda x: x[2])
  act3.sort(key = lambda x: x[2])
  act4.sort(key = lambda x: x[2])
  act5.sort(key = lambda x: x[2])
  act6.sort(key = lambda x: x[2])
  act7.sort(key = lambda x: x[2])
  print(solve(act))
  print(phi(act, 0, []))
  print(solve(act1))
  print(phi(act1, 0, []))
  print(solve(act2))
  print(phi(act2, 0, []))
  print(solve(act3))
  print(phi(act3, 0, []))
  print(solve(act4))
```

```
  print(phi(act4, 0, []))
  print(solve(act5))
  print(phi(act5, 0, []))
  print(solve(act6))
  print(phi(act6, 0, []))
  print(solve(act7))
  print(phi(act7, 0, []))

main()
```

Salida

```
[(2, 25), (24, 28), (3, 23)]
iterativo = ([(24, 28)], 1)
recursivo = 1
[(18, 30), (21, 28), (3, 14), (13, 19)]
iterativo = ([(13, 19), (21, 28)], 2)
recursivo = 2
[(1, 3),(4, 20),(21, 26)]
iterativo = ([(1, 3), (21, 26)], 2)
recursivo = 2
[(16, 20), (5, 10), (12, 35), (18, 30), (21, 28), (3,14), (13, 19)]
iterativo = ([(16, 20), (5, 10), (21, 28)], 3)
recursivo = 3
[(2, 26), (24, 28), (3, 23)]
iterativo = ([(24, 28)], 1)
recursivo = 1
[(13, 14), (15, 16), (23, 37)]
iterativo = ([(13, 14), (15, 16), (23, 37)], 3)
recursivo = 3
[(17, 26), (2, 3), (18, 20), (20, 22)]
iterativo = ([(2, 3), (18, 20), (20, 22)], 3)
recursivo = 3
[(4,27), (10,16), (21,4)]
iterativo = ([(21, 4), (10, 16)], 2)
recursivo = 2
```